Manual De Uso.

Una tecnología que debe ser compartida.

INTRODUCCION

Se ha presentado un código en Python que puede servir de eje para poder explicar las estructuras que resuelven estructuras conocidas. Es por ello que se va a presentar distintos códigos, y se va a explicar cómo usarlos y cuáles fueron sus filosofías a la hora de diseñarlos. Así como sus debilidades.

Todos los ficheros han sido compilados, o son interpretados, para Python 3.x. Esto quiere decir que para su aprovechamiento es aconsejable instalar ese paquete, que es gratuito. Una vez instalado, desde el browser (indicador de comandos) se podrá aprovechar las estructuras mediante importaciones con el comando **import** según corresponda.

Todo este software es de libre distribución y se le puede añadir interfaces, hacer modificaciones, etcétera..., se ruega que no se olviden de quién es el autor original por si vale la pena referenciarlo; el verdadero objeto del autor es obtener un reconocimiento y poder hacer conferencias o talleres para desarrollar esta tecnología.

1. CÁLCULO DEL ÍNDICE DE HOSOYA. TÉCNICA DE LA SALVAGUARDA.

De todos los problemas que se van a exponer, el de resolución más compleja es el índice de Hosoya, esto es, calcular el número de MATCH's que tiene un grafo. Contar el número de MATCH's equivale en complegidad a calcular el inmanente de una matriz, o también saber cuántas soluciones satisfacen una fórmula booleana. Es decir, si partimos de un grafo y lo convertimos rápidamente en su matriz característica, el cálculo del inmanente nos daría el número de MATCH's, y lo mismo pasaría si diéramos con la fórmula booleana para contar el número de casos.

Al problema de contar el número de casos se le llama Sharp y, en esta muestra de código, vamos a evitar la transformación del problema en otros más sencillos. Aquí procederemos a explicar en qué consiste la técnica de la salvaguarda.

A través del patrón estado, nuestro sistema almacenará una pregunta que no sabe responder y, si una vez resuelta, vuelve a preguntar lo mismo entonces aprovechará la solución que almacenó. Esto, en definitiva, en la idea de que muchas operaciones se basan en un factor común que se repite, es en lo que se fundamenta la técnica que se va a exponer.

1.1 Forma de Uso

Fichero: Hosoya.py

grafoAzar (n, P)	Devuelve un grafo simétrico al azar con n vértices bajo una probabilidad P (sobre 1) a que dos vértices se relacionan.	> grafoAzar (4, 0.5) > grafoAzar (10, 0.1)
rend2 (m, n, P, olvida=True, muestraLote=False)	Genera m grafos diferentes de n vértices con una probabilidad P a que dos vértices se relacionan y cronometra cuánto tarda para generar su índice de Hosoya. También muestra cuánto espacio necesitó para almacenar estados. **olvida** si es True significa que entre intento e intento no albergará la experiencia de lo calculado anteriormente.	> rend2 (5, 10, 0.5) > rend2(5, 20, 0.1, muestraLote=True) > rend2(10, 10, 0.6, False, True)

	muestraLote si es False provocará que se devuelva la media de los resultados en vez de la muestra completa.	
testea (G)	Devuelve el tiempo en segundos en calcular el índice de Hosoya del grafo **G**.	> testea (grafoAzar(10,0.5))
Grafo(*pares)	Genera un grafo simétrico a partir de los pares introducidos como argumentos.	> G = Grafo ((1,2), (1,3))
(Grafo).clona()	Crea un clon del objeto Grafo de manera que no compartan datos.	> G2= G1.clona()
Grafo.completo(n)	Genera un grafo completo de n vértices.	> G= Grafo.completo(3)
(Grafo).__len__()	Devuelve el índice de Hosoya del Grafo.	> len (G)
(Grafo).__repr__()	Devuelve el estado del Grafo en formato cadena.	> G

1.2 Explicación del código

A la hora de crear el grafo, lo primero que hace el código es almacenar las adyacencias en un diccionario: para cada vértice almacenamos sus adyacentes; de esa manera es como trabajaremos con el grafo.

Como el objeto es hacer varias veces la misma llamada a un problema precalculado, debemos preparar el grafo de manera que esté predispuesto a precalcular para luego volver a calcular lo mismo. Para ello se procede a reordenar los vértices de manera que los más lejanos a la raiz estén lo más atrás en la lista.

Esto provocará que cuando vayamos eliminando aristas primero se eliminen los vértices que se encuentren a la cabeza, probablemente dejando invariante los vértices que se encuentren a la cola y más lejos de los primeros vértices.

Para poder *nivelar* los vértices simplemente calculamos su distancia más corta al primer vértice, se ordena y se almacenan los ciclos necesarios para hacer la ordenación. Respetando tales ciclos se permutan los vértices y ya están *nivelados*.

Una vez creado el grafo ya podemos calcular su índice de Hosoya, para ello usaremos la variable global salvaguarda, que es el diccionario donde se almacenará cada grafo al que queramos calcular su índice.

En el cálculo del índice de Hosoya existen dos casos triviales: no hay grafo, por lo que el índice es 1; si el grafo es un conjunto de caminos, entonces se puede calcular multiplicando el número de fibonacci de la longitud de cada camino. En el resto de los casos, habría que aplicar la técnica.

Lo primero que se hace, por tanto, es comprobar si ya hemos precalculado el índice, porque de ser así sólo habría que devolver el valor almacenado en la variable global.

Si no se ha calculado se procede simple y llanamente a hacer una explosión de combinaciones para sumar los distintos MATCH's posibles sobre la variable global haciendo una llamada a un clon del grafo cuyas adyacencias han sido mermadas en virtud del caso que se esté estudiando en el bucle.

1.3 Análisis no funcional.

Esta metodología funciona bien cuando los caminos son largos, sin embargo, debido a que hace una explosión combinatoria, cuando todos los caminos son muy simples (como en los grafos completos o casi completos) también dará resultados eficientes. Sin embargo no dará buenos resultados cuando se trate de caminos moderadamente cortos para una gran cantidad de vértices.

No por ello la técnica debe ser desechada, puede que si se parte de una estructura preadaptada para que trabaje bien, entonces obtendremos los resultados deseados. No hay que olvidar que varios grafos pueden tener el mismo índice de Hosoya, así como que la filosofía de la salvaguarda se puede aplicar para estructuras equivalentes sobre las que trabaje bien no funcionalmente.

Como nota adicional, huelga mencionar que la función de Fibonacci ha sido implementada mediante una función recursiva, sin prestar atención a su relación trivial con el número áureo para mejorar su rendimiento.

2. SATISFACCIÓN DE FÓRMULAS.

Los problemas de conjuntos explícitos, como se describe en el artículo adjunto, pueden ser presentados como un producto de alternancias. En el siguiente código se presenta la estructura que resuelve los problemas NP asociados a las fórmulas de la lógica booleana.

En este caso, la estructura es 100% polinomial, no puede ir ni más lento ni más rápido porque hace tres anidamientos de bucle sin condicionar. Pero se puede optimizar encriptando ligeramente el código para ofrecer servicios más concretos. Como el objeto de esta documentación es la formación, se ha elegido esta versión para su publicación.

2.1 Forma de Uso

Fichero: alternancias.py

Alt (alternancias)	Genera un objeto fórmula de alternancias a partir de una lista de listas. Los nodos últimos es preferible que sean enteros mayores que 0, porque los nodos negativos representan el literal negado. Sólo funciona adecuadamente a partir de una lista tres listas como mínimo.	> Toffoli = Alt ([[-1, 5, 6], [-2, 5, 7], [6, 7, 8], [-5, 9, 10], [-3, 9, 11], [10, 11, -4]])
(Alt).c(N, *variables)	Observa N casos (repetidos o no) que satisfacen al objeto Alt a partir de un índice N de referencia presentando en pantalla el valor de las variables mencionadas.	> Toffoli.c(16, 1, 2, 3, 4)
(Alt).m()	Muestra, para pocas cláusulas, las tablas por pantalla.	> Toffoli.m()
Alt.proyMatrices (A, B, operador=*)	Aplica un operador, que por defecto es el producto, componente a componente entre dos matrices y defuelve la matriz resultado.	> Suma = Alt.proyMatrices (A,B, lambda x,y: x+y) > X = Alt.proyMatrices(Y,Z)
Alt.prodMatrices(A, B)	Devuelve el producto de dos matrices.	> A = Alt.prodMatrices(B,C)

Alt.sharpAB (A, B, C)	Actualiza el tope de tres tablas que deben cumplir la relación de reunión.	
Alt.sharpBB (A, B, C)	Actualiza la base de tres tablas que deben cumplir la relación de reunión.	
Alt.sharpCB (A, B, C)	Actualiza el extremo de tres tablas que deben cumplir la relación de reunión.	
(Alt).clona()	Clona la estructura de tablas de una fórmula de alternancias en profundidad para mantener la misma fórmula original y sus variables.	> T2=T1.clona()
(Alt).__getitem__(item)	Devuelve el conjunto de literales que satisfacen la fórmula cuando éstas sean True, a partir de item como referencia. Si se incluye literales negativos, significa que la variable debe ser False.	> Toffoli [2] > Toffoli[1313]
(Alt).__call__(item, *variables)	Devuelve la asignación que se atribuye a cada variable a partir de item como referencia.	> Toffoli (2, 1, 2, 3, 4) > Toffoli (1313, 1, 2, 3, 4)
(Alt).filtra (*variables)	Restringe los casos de la tabla para hacer que los literales presentados se afirmen. Si el literal es negativo, entonces la variable se niega.	> Toffoli.filtra(1, -4) > Toffoli.c (10, 1,2,3,4)
(Alt).__bool__()	Nos dice si la fórmula se satisface para algún caso.	> if Toffoli: print("Satisface") > bool(Toffoli)
Alt.andeq(aux, A, B, AyB, AeqB)	Genera una lista de listas que parametrizan una fórmula de alternancias que atribuye a las variables **AyB** la fórmula **A** AND **B** **AeqB** la fórmula **A** == **B** **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	> L, aux = Alt.andeq(100, 1, 2, 3, 4) > F = Alt (L)
Alt.orAlt(aux, R, *V)	Genera una lista de listas que parametrizan una fórmula de alternancias que atribuye a la variable **R** el or de los argumentos listados. **R** = **V**[1] + **V**[2] + … **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	> L, aux = Alt.orAlt (100, 1, 2, -3, 4, -5, 6) > L, aux = Alt.orAlt(100, R, op1, op2)
Alt.andAlt(aux, R, *V)	Genera una lista de listas que parametrizan una fórmula de alternancias que atribuye a la variable **R** el and de los argumentos listados. **R** = **V**[1] & **V**[2] & … **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	> L, aux = Alt.andAlt (100, 1, 2, -3, 4, -5, 6) > L, aux = Alt.andAlt(100, R, prem1, prem2)
Alt.menor(aux, R,vars1, vars2)	Genera una lista de listas que parametrizan una fórmula de alternancias que atribuye a la variable **R** el que el listado binario **vars1** es menor que el binario **vars2**. **vars1** y **vars2** son listas que representan un natural donde el	> L, aux = Alt.menor(100, 1, [2,3,4], [5,6,7])

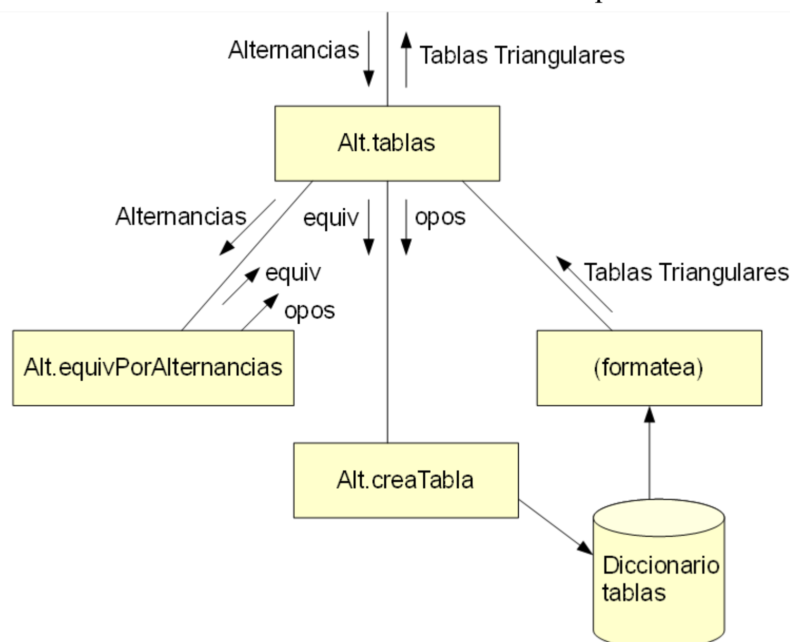
	primer elemento es la cifra menos significativa. **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	
Alt.sumAlt(aux, R, variables)	Genera una lista de listas que parametrizan una fórmula de alternancias que atribuye a la variable **R** el XOR de los literales listados en la lista **variables**. También nos devuelve la lista de literales que suman el carryOut si se hubiera hecho la suma de valores booleanos. **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	> L, cOut, aux = Alt.sumAlt(100, 1, [2,3,4])
Alt.impAlt(aux, A, *B)	Genera una lista de listas que parametrizan una fórmula de alternancias que hace que la variable A implique las alternancias de B. **A** implica (**B**[0] | **B**[1] | …) **aux** es el número de variable supremo con el que aún no se ha trabajado, y devuelve el nuevo valor para futuros usos de **aux**.	>L, aux = Alt.impAlt(100, 1, 2,3,4 5)

1.2 Explicación del código

A la hora de explicar las distintas funciones que componen el sistema Alt, huelga mencionar cuatro grandes subsistemas: la **inicialización**, el **afilamiento** y las **observaciones** y los **usos**. Con el fin de estructurar mejor las explicaciones, se procederá a exponer cada parte en epígrafes diferentes.

1.2.1 Inicialización

La inicialización comprende la función que lleva a cabo **Alt.tablas(Alternancias)**, que genera, a partir de una lista de alternancias de literales una lista de tablas que relaciona cada par de cláusulas.



Inicialmente mediante la función Alt.equivPorAlternancias se extraen dos listas: la lista de equivalencias **equiv** y la lista de opuestos **opos**, que contendrán pares de posiciones de literales en cláusulas que o bien son iguales en distintas cláusulas o bien son opuestos, esto es, de signo contrario.
Con esa información se va llamando a Alt.creaTabla que irá actualizando un diccionario de tablas cuya clave es un par de posiciones dentro de la lista de cláusulas. De esta manera se establece la coherencia más básica para cada par de cláusulas y, mediante un formateo de la salida, se convierte en una lista de matrices.

1.2.2 Afilamiento

El proceso de afilamiento consiste, tal como se ha implementado, en hacer dos barridos: directo e inverso, de todas las tablas llamando a la función **self._afila (I, J)**, que lo que hace es actualizar la tabla que relaciona la cláusula I con la cláusula J en la fórmula. Para llevar a cabo el proceso de actualización se ocupará de hacer uso de la función Alt.sharpAB, Alt.sharpBB o Alt.sharpCB según corresponda al combinar las coordenadas I y J con una tercera K que forme una trinca.

Al proceso se le llama afilamiento por el nombre en inglés sharp, que consiste en ser mordaz o ser más agudo, cosa que se pretende hacer con la cuña con el fin de acercarse a la resolución sharp, que es para contar el número de casos.

Sin embargo, la resolución del afilamiento será una tabla donde se asegurará que allá donde en una celda haya un 1, reconocerá al menos una solución para ese par de literales en sus correspondientes cláusulas dentro de la fórmula.

1.2.3 Observaciones

Partiendo del hecho de que la tabla está afilada, podemos asegurar que cuando accedamos a cualquier tabla y encontremos un 1, al filtrar los resultados mediante la función **self.filtra(*variables)**, seguiremos teniendo una tabla con matrices no nulas.

La función **self.filtra(*variables)** llama a la función **self._posicionesVecinas(*variables)** para saber qué literales deben anularse, si procediera por el signo de cada literal. Acto seguido se procede a anular el literal dentro de su cláusula considerando si se debe anular una fila o una columna, y para anular ya sea una fila o una columna usaremos la función **self._anulaTabla(I, J, col)**.

La función **self.__getitem__(item)**, que es la notación que usa Python para definir el manejo de los corchetes con el objeto, se ocupa de seleccionar inicialmente una solución desde la primera tabla (la tabla principal), para obtener el primer par de literales para la solución a partir de la elección adoptada por **item** dentro del número de soluciones posibles que ofrecía la tabla. Acto seguido, se va llamando a la función **self._trasiego(k, *Asig)** donde Asig nos dice a cuántas cláusulas le hemos dado solución y, por tanto, nos desvela el número de cláusula sobre la que se trabaja y k nos dice el literal que estudiamos en tal cláusula. El objeto de tal función es decirnos si para las soluciones arrastradas la asignación para la cláusula actual del literal k, si tiene o no solución, tras comprobar en todas las tablas que se asocia con un 1.

De esta manera se obtiene el listado de posiciones que se traducirá en los nombres de los literales.

La función **self.__call__(item, *variables)** es la notación que usa Python para las llamadas a función y para que nos devuelva los valores de verdad que necesitan las variables para satisfacer el caso item se usará la función **self.__getitem__(item)**.

La función de llamada recoge la solución y llama a la función **self.__formatoSolucion(variable, solucion)**, que lo único que hace es aplicar una lógica abierta o cerrada en virtud de cómo se definió el literal dentro de la fórmula. Obviamente, y por simplificar, es posible eliminar los literales negativos de estos esquemas, y nos ahorraríamos mucho código, pero también iría el sistema más lento al tener que incluir cláusulas que simulen la oposición de literales.

1.2.4 Usos

Entre los usos posibles, cabe destacar la estructura más sencilla que debe ser usada para encontrar la equivalencia entre las fórmulas de la lógica y las de las alternancias. Esta es la función **Alt.andeq (aux, A, B, AyB, AeqB)**. La justificación de porqué esta función nos devuelve lo que se asegura es mejor probarlo por casos, ya que son sólo cuatro combinaciones y la corroboración de que cualquier otra combinación no es posible.

En estas funciones la variable **aux** funcionará como valor supremo y siempre deberá ser superior al resto de las variables con las que se trabaja antes de llamar a la función que genera la lista. De esta manera la función que genera la lista de alternancias usará nuevas variables a partir de **aux** y devolverá cuál es el nuevo valor supremo.

De la misma manera se puede entender el uso de las funciones **Alt.orAlt** y **Alt.andAlt** pues, como si fuera un proceso de definir un sistema digital, poco a poco se van construyendo componentes más complejas y, si se quiere multiplicar varios literales o hacer un sumatorio booleano es interesante tener esas funciones a mano.

Una función también fácil de entender puede ser **Alt.impAlt (aux, A, *B)** debido a que será útil a la hora de determinar si un grafo es subgrafo de otro.

Mientras que **Alt.sumAlt** y **Alt.menor** están pensados para generar los casos del problema **BPP** (determinación de la mínimas particiones necesarias que hacen que sus elementos no sumen más que una cantidad prefijada), lo cual, si lo menciono, es porque es fácil de ver cómo también se encuentra entre los problemas **P**, ya que es polinomialmente derivable al manejo de estas funciones. Esta afirmación que acabo de verter contradirá las espectativas de muchas personas que esperaban en ese problema algo descomunalmente complejo, cuando en realidad no lo es especialmente más que el cálculo del índice de Hosoya y, al mismo tiempo, dentro de los problemas NP podría ser el más *pesado* y *lento*.

3. PROBLEMAS DE SUBGRAFOS. SI EXACTOS, PESADOS Y LENTOS.

Una forma de uso que le podemos dar a la librería de alternancias es para resolver los problemas de subgrafos o distintos tipos de puzzles haciendo uso de la lógica. Desde aquí no se recomendará la formulación lógica exacta de esos problemas ya que los resultados, a pesar de que no irán más allá

de una complejidad cúbica, en un ordenador convencional en solitario se puede convertir en una epopeya. Huelga mencionar la existencia de una gran cantidad de algoritmos de tirada exponencial que han sido diseñados partiendo de una filosofía orientada por el patrón alarma, éstos funcionan inesperadamente mejor de lo que muchos esperarían, aunque no aseguren al 100% un resultado fiable, cuando sí un resultado automático. No dudaré en publicar tales máquinas si así se requiriera, al mismo tiempo que existen mecanismos que aproximan para la mayoría de los grafos resultados bastante potables que dan una respuesta en poco tiempo.

Sin embargo, y como objeto de estudio para tener una apreciación completa, se presenta también un algoritmo que aprovecha la nomenclatura de los grafos para resolver el problema de saber si un subgrafo está contenido en otro.

Este problema, bien aprovechado, es capaz de resolver otros como la determinación de si un grafo tiene un camino hamiltoniano o la determinación de cuál es el camino más largo en un grafo.

3.1 Forma de Uso

Fichero: grafos.py

Grafo (arg)	Devuelve un grafo simétrico a partir de una lista de pares.	> G = Grafo ([(1,2), (2,3), (3,4), (4,1)]) > H= Grafo([(1,2),(2,3),(3,4)])
(Grafo).__getitem__(item)	Te devuelve la lista de adyacentes del vértice item en el grafo.	> G[1]
Grafo.subGraphProblem(G1, G2)	Genera una lista de alternancias que equivalen a resolver si el grafo G2 está contenido en G1.	> L , aux = Grafo.subGraphProblem(G, H) > T = Alt(L) #pesado

4. CONTEO DE FÓRMULAS SIMPLES. TETRALÓGICA.

Otro arreglo filosófico que nos ayuda a aproximarnos al conteo del número de casos es una técnica que desarrollé hace muchos años, a la que llamé tetralógica: consistía en hacer uso de una lógica de cuatro valores y almacenar todos los posibles valores de verdad de una expresión aprovechando el empaquetamiento que ofrece el valor *Indeterminado* que, en mi lógica era el número 3.

Este tipo de esquemas pueden sucumbir a una explosión combinatoria con respecto el número de variables cuando no con respecto al número de cláusulas. Sin embargo, entrando en matices, aunque nos ofreciera una resolución dentro de una cota polinomial, en la práctica podríamos encontrar fórmulas donde el coeficiente por el que se eleva la variable fuera demasiado grande, demasiado pesado, como para que fuera práctico.

Es por ello que estas técnicas pueden tener un uso práctico para las fórmulas que no usen un número demasiado grande de variables o cuando las cláusulas estén muy interconectadas entre ellas al tener muchos literales que podrían oponerse.

Estas estructuras eran teorizadas en mi primer libro, pero huelga mencionar la estructura que vamos a usar aquí, principalmente, para acabar con el mito del coNP.

Gracias a la estructura arbórea que se va a explicar en esta epígrafe, podremos comprobar cómo una fórmula bien formada puede ser negada y, acto seguido, no perder eficiencia a la hora de ser evaluada; todo lo más, veremos otra fórmula compuesta por alternancias, o con lo que precisemos, para llevar a cabo las operaciones pertinentes.

Por tanto, si bien la lógica de los cuantificadores nos llevaría al problema de la *pesadez* que mencioné al principio de este apartado, la sencillez de demostración de un problema que sea expresable en el álgebra de Bool, implicará que sea igual de sencillo demostrar su opuesto y, por tanto, es factible en **P-ligero** deducir si una fórmula es un teorema del álgebra de Bool.

4.1 Forma de uso

Fichero: arbolsharp.py

Literal(cadena)	Genera un literal de lógica simbólica que podrá estar negado haciendo uso del carácter '-'. Los literales suelen tener el formato [signo]cadena[numero]	> P = Literal ('-P') > Literal ('V23')
(Literal).__neg__()	Le cambia el signo al literal, al generar su opuesto.	> noP = -P
ArbolNAOI(arbol)	Genera un arbol de derivación que se vale de funciones NOT, AND, OR e IMPLICACIÓN. Su primitiva es un objeto Literal.	> A = ArbolNAOI (Literal('p'))
(ArbolNAOI).__and__(other)	Genera un objeto ArbolNAOI a partir de aplicar AND sobre otro arbol.	> A & B
(ArbolNAOI).__or__(other)	Genera un objeto ArbolNAOI a partir de aplicar AND sobre otro arbol.	> A | B
(ArbolNAOI).__neg__()	Genera un objeto ArbolNAOI a partir de aplicar NOT sobre el arbol.	> -A
(ArbolNAOI).__gt__(other)	Genera un objeto ArbolNAOI a partir de aplicar la IMPLICACIÓN sobre otro arbol.	> A > B
sharp(propuesta, nvariables=None)	Devuelve el número de casos que valida al objeto ArbolNAOI. Se puede especificar el número de literales que hay en la fórmula para evitar un recorrido.	> sharp (A>B)
(ArbolNAOI).toAsignaciones(**auxiliares=0,** **prefijo='z')**	Transforma el objeto ArbolNAOI en un producto de asignaciones de la forma (literal = literal & literal) donde cada literal es una fórmula atómica que puede ser negada. Además, los literales auxiliares se irán creando a partir de los parámetros fijados con el formato prefijo+número.	

5. MECANISMOS ANALÍTICOS

Por último mencionaré que, aunque no lo desarrollo en estos documentos, también habré desarrollado mecanismos que aprovechan trasformadas y otras señales que nos permiten obtener cálculos que, mediante conjuntos explícitos, suelen ser difíciles de conseguir. Si bien, el número de casos se puede asociar al cálculo del área de una función, o también el cálculo del periodo de una función acaba siendo un problema de encontrar el cero de su polinomio característico a través de un desarrollo de Taylor, todas esos mecanismos deben ser correctamente complementados con un estudio preliminar que permita ajustar el modelo transformado con el problema de origen.
Estos problemas de desajustes nos lleva a un sinfín de enfoques varios y una lucha completamente análoga a lo que se ha presentado en esta documentación y es, por tanto, objeto de futuras entregas.